University of Pittsburgh

Named Functions

Week 4

MEMS 1140—Introduction to Programming in Mechanical Engineering

Learning Objectives (L.O.)

At the end of this lecture, you should understand/be able to:

- ❏ What named functions are;
- ❏ How named vs. anonymous functions differ;
- ❏ Write named functions;
- ❏ Determine when to use a named function;
- ❏ Apply a named function to the cross product.

Table of Contents (ToC)

1 – What is a Named Function

Named functions encapsulate one or more operations to be extracted from the main body of code.

They are most often defined in a standalone file and can be called from the command window and from scripts.

Named functions are best utilized for modularizing a larger portion of code than anonymous functions.

2 – Anonymous Vs. Named Functions

Anonymous functions: stored as a variable within a script.
Named functions: stored as a standalone file.

Anonymous functions: only encompass one line of code.
Named functions: can encompass infinite lines of code.

Anonymous functions: intended for simple operations.
Named functions: used for complex, multi-step operations.

3 – Function Definition Keywords

The definition of a named function is wrapped with the **`function`** and **`end`** keywords:

```
function

end
```

Named Function (.m)

3 – Function Inputs

The **inputs**, function name, and outputs are then specified with the following structure:

```
function [<outputs>] = functionName(<inputs>)

end
```

Named Function (.m)

3 – Assigned Function Name

The inputs, **function name**, and outputs are then specified with the following structure:

```
function [<outputs>] = functionName(<inputs>)

end
```

Named Function (.m)

3 – Function Outputs

The inputs, function name, and **outputs** are then specified with the following structure:

```
function [<outputs>] = functionName(<inputs>)

end
```

Named Function (.m)

3 – Encapsulated Code

Finally, the operations encapsulated by the function are written in the middle:

```
function [<outputs>] = functionName(<inputs>)
  <encompassed code>
end
```

Named Function (.m)

3 – Variable Scope

The body of a function only has access to variables passed through the inputs or defined in the body itself.

Similarly, code outside the function can only access variables defined within the body if they are returned via brackets.

This is called **variable scope**.

It is important to consider variable scope when you work with functions.

3 – A Simple Example

Consider a function that accepts two numbers as inputs and prints all of the arithmetic operations between them:

```
function [] = printArithmetic(a, b)
   .
   .
   .
   .
   .
end
```

Named Function (.m)

3 – A Simple Example

Consider a function that accepts two numbers as inputs and prints all of the arithmetic operations between them:

```
function [] = printArithmetic(a, b)
  fprintf('a + b = %f \n', a + b)    % \n is the newline character
  .
  .
  .
  .
end
```

Named Function (.m)

3 – A Simple Example

Consider a function that accepts two numbers as inputs and prints all of the arithmetic operations between them:

```
function [] = printArithmetic(a, b)
  fprintf('a + b = %f \n', a + b)
  fprintf('a - b = %f \n', a - b)
  .
  .
  .
end
```

Named Function (.m)

3 – A Simple Example

Consider a function that accepts two numbers as inputs and prints all of the arithmetic operations between them:

```
function [] = printArithmetic(a, b)
  fprintf('a + b = %f \n', a + b)
  fprintf('a - b = %f \n', a - b)
  fprintf('a * b = %f \n', a * b)
  .
  .
end
```

Named Function (.m)

3 – A Simple Example

Consider a function that accepts two numbers as inputs and prints all of the arithmetic operations between them:

```
function [] = printArithmetic(a, b)
  fprintf('a + b = %f \n', a + b)
  fprintf('a - b = %f \n', a - b)
  fprintf('a * b = %f \n', a * b)
  fprintf('a / b = %f \n', a / b)
  .
end
```

Named Function (.m)

3 – A Simple Example

Consider a function that accepts two numbers as inputs and prints all of the arithmetic operations between them:

```
function [] = printArithmetic(a, b)
  fprintf('a + b = %f \n', a + b)
  fprintf('a - b = %f \n', a - b)
  fprintf('a * b = %f \n', a * b)
  fprintf('a / b = %f \n', a / b)
  fprintf('a ^ b = %f \n', a ^ b)
end
```

Named Function (.m)

3 – Calling a Named Function

To use this function, it can be called by its name and passed two inputs as follows:

```
>> printArithmetic(3,4)
a + b = 7.000000
.
.
.
.
```

Command Window

3 – Calling a Named Function

To use this function, it can be called by its name and passed two inputs as follows:

```
>> printArithmetic(3,4)
a + b = 7.000000
a - b = -1.000000
.
.
.
```

Command Window

3 – Calling a Named Function

To use this function, it can be called by its name and passed two inputs as follows:

```
>> printArithmetic(3,4)
a + b = 7.000000
a - b = -1.000000
a * b = 12.000000
.
.
```

Command Window

3 – Calling a Named Function

To use this function, it can be called by its name and passed two inputs as follows:

```
>> printArithmetic(3,4)
a + b = 7.000000
a - b = -1.000000
a * b = 12.000000
a / b = 0.750000
.
```

Command Window

3 – Calling a Named Function

To use this function, it can be called by its name and passed two inputs as follows:

```
>> printArithmetic(3,4)
a + b = 7.000000
a - b = -1.000000
a * b = 12.000000
a / b = 0.750000
a ^ b = 81.000000
```

Command Window

4 – Appropriate Applications

Named functions — just like anonymous functions — are used to extract repeated operations or to help make code more readable and/or simpler to debug.

But unlike anonymous functions, they can encapsulate more than one line of code.

This is useful for extracting code that involves multiple processing steps like the example given previously.

5 – Cross Production Function

Recall the cross product between two vectors:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \hat{\imath} & \hat{\jmath} & \hat{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

$$\vec{a} \times \vec{b} = (a_y b_z - a_z b_y)\,\hat{\imath} - (a_x b_z - a_z b_x)\,\hat{\jmath} + (a_x b_y - a_y b_x)\,\hat{k}$$

Algebra like this is prone to errors and becomes tedious to write repeatedly. A function is perfect for this.

5 – Function Wrapper Definition

Start out by writing the function definition wrapper:

```
function output_vector = crossProduct(v1, v2)
   .
   .
   .
   .
end
```

Named Function (.m)

5 – Compute the X-Component

Then, compute the **x**-component of $\vec{v}_1 \times \vec{v}_2$:

```
function output_vector = crossProduct(v1, v2)
  x_component = (v1(2) * v2(3)) - (v1(3) * v2(2));
   .
   .
   .
end
```

Named Function (.m)

5 – Compute the Y-Component

Then, compute the **y**-component of $\vec{v}_1 \times \vec{v}_2$:

```
function output_vector = crossProduct(v1, v2)
  x_component = (v1(2) * v2(3)) - (v1(3) * v2(2));
  y_component = -( (v1(1) * v2(3)) - (v1(3) * v2(1)) );
  .
  .
end
```

Named Function (.m)

5 – Compute the Z-Component

Then, compute the **z**-component of $\vec{v}_1 \times \vec{v}_2$:

```
function output_vector = crossProduct(v1, v2)
  x_component = (v1(2) * v2(3)) - (v1(3) * v2(2));
  y_component = -( (v1(1) * v2(3)) - (v1(3) * v2(1)) );
  z_component = (v1(1) * v2(2)) - (v1(2) * v2(1));
  .
end
```

Named Function (.m)

5 – Compile the Output

Finally, package all three components into **`output_vector`**:

```
function output_vector = crossProduct(v1, v2)
  x_component = (v1(2) * v2(3)) - (v1(3) * v2(2));
  y_component = -( (v1(1) * v2(3)) - (v1(3) * v2(1)) );
  z_component = (v1(1) * v2(2)) - (v1(2) * v2(1));
  output_vector = [           ,           ,           ];
end
```

Named Function (.m)

5 – Compile the Output

Finally, package all three components into **`output_vector`**:

```
function output_vector = crossProduct(v1, v2)
  x_component = (v1(2) * v2(3)) - (v1(3) * v2(2));
  y_component = -( (v1(1) * v2(3)) - (v1(3) * v2(1)) );
  z_component = (v1(1) * v2(2)) - (v1(2) * v2(1));
  output_vector = [x_component,            ,            ];
end
```

Named Function (.m)

5 – Compile the Output

Finally, package all three components into **`output_vector`**:

```
function output_vector = crossProduct(v1, v2)
  x_component = (v1(2) * v2(3)) - (v1(3) * v2(2));
  y_component = -( (v1(1) * v2(3)) - (v1(3) * v2(1)) );
  z_component = (v1(1) * v2(2)) - (v1(2) * v2(1));
  output_vector = [x_component, y_component,            ];
end
```

Named Function (.m)

5 – Compile the Output

Finally, package all three components into **`output_vector`**:

```
function output_vector = crossProduct(v1, v2)
  x_component = (v1(2) * v2(3)) - (v1(3) * v2(2));
  y_component = -( (v1(1) * v2(3)) - (v1(3) * v2(1)) );
  z_component = (v1(1) * v2(2)) - (v1(2) * v2(1));
  output_vector = [x_component, y_component, z_component];
end
```

Named Function (.m)

5 – Cross $\hat{i} \times \hat{j}$

To test this function, let's use the three unit direction vectors.

```
>> i = [1, 0, 0]; j = [0, 1, 0]; k = [0, 0, 1];
>> .
.
.
```

Command Window

5 – Cross $\hat{i} \times \hat{j}$

To test this function, let's use the three unit direction vectors.

```
>> i = [1, 0, 0]; j = [0, 1, 0]; k = [0, 0, 1];
>> crossProduct(i,j)     % should produce k
.
.
```

Command Window

5 – Cross $\hat{i} \times \hat{j}$

To test this function, let's use the three unit direction vectors.

```
>> i = [1, 0, 0]; j = [0, 1, 0]; k = [0, 0, 1];
>> crossProduct(i,j)     % should produce k
ans =
     0     0     1
```

Command Window

5 – Cross $\hat{j} \times \hat{k}$

To test this function, let's use the three unit direction vectors.

```
>> i = [1, 0, 0]; j = [0, 1, 0]; k = [0, 0, 1];
>> .
.
.
```

Command Window

5 – Cross $\hat{j} \times \hat{k}$

To test this function, let's use the three unit direction vectors.

```
>> i = [1, 0, 0]; j = [0, 1, 0]; k = [0, 0, 1];
>> crossProduct(j,k)     % should produce i
.
.
```

Command Window

5 – Cross $\hat{j} \times \hat{k}$

To test this function, let's use the three unit direction vectors.

```
>> i = [1, 0, 0]; j = [0, 1, 0]; k = [0, 0, 1];
>> crossProduct(j,k)     % should produce i
ans =
     1     0     0
```

Command Window

5 – Cross $\hat{k} \times \hat{i}$

To test this function, let's use the three unit direction vectors.

```
>> i = [1, 0, 0]; j = [0, 1, 0]; k = [0, 0, 1];
>> .
.
.
```

Command Window

5 – Cross $\hat{k} \times \hat{i}$

To test this function, let's use the three unit direction vectors.

```
>> i = [1, 0, 0]; j = [0, 1, 0]; k = [0, 0, 1];
>> crossProduct(k,i)     % should produce j
.
.
```

Command Window

5 – Cross $\hat{k} \times \hat{i}$

To test this function, let's use the three unit direction vectors.

```
>> i = [1, 0, 0]; j = [0, 1, 0]; k = [0, 0, 1];
>> crossProduct(k,i)     % should produce j
ans =
     0     1     0
```

Command Window

6 – Summary

This lecture covered:

- ✓ What named functions are

  Named functions have large bodies of code to modularize complex portions of code.

- ✓ How named and anonymous functions differ

  Anonymous: stored in a variable, one line of code, used for simple operations. Named: stored as `.m` files, infinite lines, used for complex operations.

6 – Summary

✓ L.O.1
✓ L.O.2
✓ L.O.3
✓ L.O.4
✓ L.O.5

✓ How to write named functions

The syntax includes the **`function`** and **`end`** keywords as well

```
<output> = functionName(<input>)
```

✓ How to determine when to use a named function

They are used for extracted repeated complex operations. In particular, when a bit of code requires several steps to complete.

6 – Summary

✓ How to apply a named function to the cross product

After defining the function inputs, outputs, and its name, the code solves for each component of the output vector as distinct steps and then combines each component into one vector.